

Intergiciels, composants et architectures logicielles

Université des Sciences et Technologies de Lille
Master Recherche Informatique

L. Duchien, C. Gransart, A.-F. Le Meur, P. Merle, R. Rouvoy,
L. Seinturier (resp.)

USTL/LIFL/INRIA -Équipe-projet INRIA ADAM



Planning des cours

2

Architecture logicielle (L. Duchien, A.-F. Le Meur)

📅 mercredi 17,24/9 - 1/10- 10h30-12h

Composant (L. Seinturier)

📅 mercredi 8/10, 15/10 - 10h30-12h

Intergiciel (P. Merle)

📅 mercredi 22/10, 5/11 - 10h30-12h

Middleware for Dependability and Adaptation (R. Rouvoy)

📅 mercredi 12/11 & 21/11 - 10h30-12h

Application mobile (C. Gransart)

📅 mercredi 26/11 - 10h30-12h



Contenu et modalités

3

Modalités d'évaluation

- ☞ fiche de lecture **individuelle** 40%
 - ➔ *présentation du sujet mercredi 24/9*
 - ➔ *à lire : document sur la théorie de l'adaptation -issu du projet européen MUSIC : <http://www.intermedia.uio.no/display/madam/D2.2+-+Theory+of+Adaptation>*
 - ➔ *à écrire : 5-10 pages*
 - résumer le document
 - recherche bibliographique sur les travaux connexes
 - ➔ *à rendre pour le 30/11/2008*
 - par mail : Laurence.Duchien@inria.fr
- ☞ examen final en décembre (10/12) 60%



4

**Modèles Abstraites à Composants et
Langage de Description
d'Architectures
(ADLs)**
Systèmes de composition de boîtes noires

Laurence Duchien
Equipe-Projet ADAM
INRIA- USTL-LIFL
<http://adam.lille.inria.fr/>

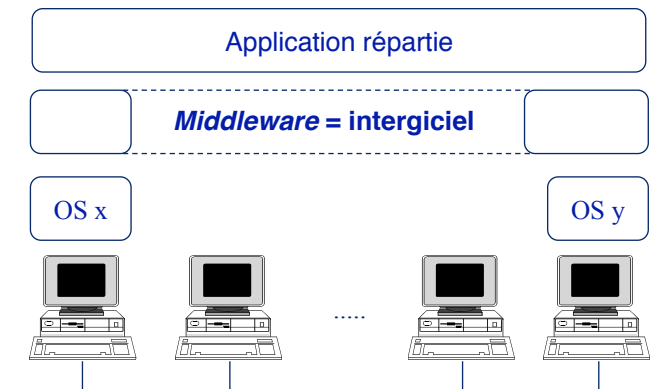


1. Introduction

5

« *Middleware is everywhere* »

© IBM



1. Introduction

6

Dualité application / intergiciel (*middleware*)

Middleware

s'appuie sur les API de « haut niveau » fournis par l'OS
fournit des API de « haut niveau » aux applications réparties
masque l'hétérogénéité des OS sous-jacents
fournit des protocoles de communication distants
fournit des services aux applications réparties
fournit un interfaçage avec 1 ou plusieurs langages de progr.



1. Introduction

7

Dualité application / intergiciel (*middleware*)

Concepts du génie logiciel
procédure, objet, composant, ...
appliqués aux 2 niveaux
parfois de manière uniforme

☞ ex. : appli objet sur *middleware* objet (Java RMI, CORBA/C++)

parfois en décalage

☞ ex. : appli composants sur *middleware* objet (EJB, CCM)

☞ ex. : appli C sur *middleware* objet (CORBA)



Sommaire

8

- I Définitions générales
- II Notion d'architecture à composants
- III Modèles et langages à composants
 - 1 Modèles à composants académiques
 - 2 ADLS classiques
 - 3 Les standards
- IV Un exemple d'évolution d'architecture
- V Conclusion



I. Définitions générales



Définitions

Origine des architectures logicielles « explicites »

- ☞ Les difficultés rencontrées par les concepteurs de gros logiciels
- ☞ Les caractéristiques de la programmation à grande échelle
- ☞ Les besoins de **réutilisation** de logiciel

Définition ANSI/IEEE 1471

- ☞ Une architecture logicielle est l'organisation fondamentale d'un système incorporée dans ses composants, les relations entre ses composants et leur lien avec leur environnement, et les principes qui régissent sa conception et son évolution

Définition d'une architecture logicielle [Shaw & Garlan]

- ☞ Architecture logicielle = niveau de conception
 - *description des éléments à partir desquels le système est construit*
 - *interactions entre ces éléments*
 - *patrons/styles qui guident leur composition*
 - *les contraintes sur ces patrons/styles*



Définitions

11

Objectif d'une architecture logicielle

- ☞ définir la structure du système,
- ☞ faire correspondre les besoins utilisateurs et l'implantation
- ☞ comprendre les préoccupations d'un système à différents niveaux (vue globale des flux, structure de contrôle, dimensionnement, etc)
- ☞ réduire les coûts de développement (maîtriser et réutiliser)



Définitions

12

Rôle d'une architecture logicielle

- ☞ L'analyse : raisonner sur la qualité du système qui va être implémenté (perfs, maintenabilité, vérification de cohérence, de complétude, de correction)
- ☞ La compréhension : rendre explicite les décisions architecturales
- ☞ La réutilisation : patrons architecturaux - descriptions de certaines parties réutilisables
- ☞ l'évolution : gestion au mieux de la propagation des changements et évaluation des coûts de l'évolution



II. Notion d'architecture à composants



Notion d'architecture

Architecture : modélisation du système sous la forme de

- ☞ d'éléments représentant les unités de calcul et de sauvegarde des données :
composants ou modules
- ☞ d'éléments représentant les interactions entre les éléments précédents :
connecteurs
- ☞ des liens entre les deux types d'éléments précédents et leur organisation:
assemblage ou composition
- ☞ des **contraintes** représentant les choix de conception

- ☞ l'organisation peut être :
 - **hiérarchique** : un élément architectural donné (composant ou connecteur) peut être décrit à l'aide d'autres éléments
 - **plate** : un élément architectural donné est une boîte noire. Il ne peut pas être décrit en fonction d'autres éléments



Notion d'architecture

15

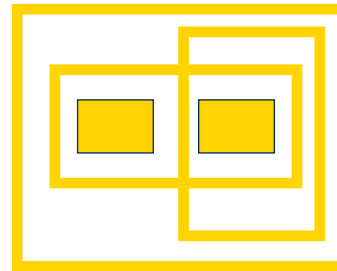
Exemple de composant composite dans l'ADL Fractal

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="Main" />
  </interface>
  <component name="Client">
    <interface name="r" role="server" signature="Main" />
    <interface name="s" role="client" signature="Service" />
    <content class="ClientImpl" />
  </component>
  <component name="Server">
    <interface name="s" role="server" signature="Service" />
    <content class="ServerImpl" />
  </component>
  <binding client="this.r" server="client.r" />
  <binding client="client.s" server="server.s" />
</definition>
```

membrane



contenu



À quel niveau du cycle de développement sommes-nous ?

16

- ↳ le contrôle de la complexité du logiciel
 - ↳ *représentation abstraite*
- ↳ maîtriser l'**architecture conceptuelle** d'un système
 - ↳ *la modification d'un système complet est très difficile*
 - ↳ *la modification d'un « plan » est plus simple*
- ↳ L'architecture logicielle correspond au « plan » d'un logiciel selon un point de vue
 - ↳ *prend en compte la complexité*
 - ↳ *augmente les possibilités de réutilisation des composants du marché*
 - ↳ *permet l'utilisation de méthodes formelles sur des parties précises de l'architecture*

Principales difficultés : complexité, conformité, transparence, interopérabilité



Traitement actuel des archi. log.

17

Compréhension au niveau de l'intuition, de l'expérience, de l'anecdote,...
Peu de méthodes ou de langages dans les milieux industriels

☞ Description informelle

- ➔ *boxologie avec une sémantique floue*
- ➔ *prose informelle*
- ➔ *au mieux définition des interfaces des différents modules (IDL?)*

☞ Cependant on y associe un vocabulaire riche d'un point de vue sémantique

- ➔ *RPC, Client/serveur, composant, couche, répartition, OO,...*



Une règle de base de conception

18

☞ L'abstraction : Se focaliser sur un problème à la fois et oublier les autres

☞ *Abstraction is neglection of unnecessary detail*

- ➔ *Considérer uniquement les informations essentielles*

☞ Séparation des préoccupations

- ➔ *Les concepts doivent être séparés -> ils peuvent être spécifiés de façon indépendante*
- ➔ *Spécifications dimensionnelles*
- ➔ *Spécifier différents points de vue*

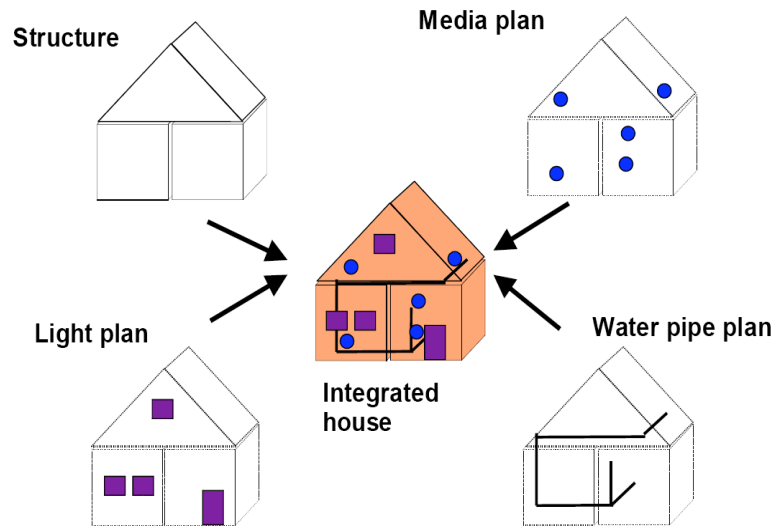
☞ Un exemple de SoC

- ➔ *Mécanisme :*
 - La réalisation technique d'une solution
- ➔ *Politique*
 - La paramétrisation de la solution
- ➔ *Séparer mécanisme et politique : leur permettre d'évoluer indépendamment*



Préoccupation dans une architecture

19



 INRIA centre de recherche
LILLE - NORD EUROPE



Les points de vues de Kruchten : 4+1

20

4 + 1 :

- ↳ **Vue logique** : spécification des besoins fonctionnels d'un point de vue comportemental (diagramme de classe et d'objets)
- ↳ **Vue de processus** : les objets sont projetés en processus - aspects non fonctionnels liés à la distribution, la concurrence, la tolérance aux pannes et l'intégrité du système
- ↳ **Vue de développement** : organisation des modules et des bibliothèques dans le processus de développement
- ↳ **Vue physique** : projection des autres éléments aux noeuds de traitement et de communication (qualité de performance et de disponibilité)
- ↳ **Vue « plus un »** : les autres vues sont projetées entre elles. Permet de voir les interactions



 INRIA centre de recherche
LILLE - NORD EUROPE



Construire une architecture

21

Deux façons de concevoir une architecture

- ☞ par décomposition d'un système en un ensemble d'éléments le constituant
- ☞ par composition d'un système à partir d'éléments (existants)

Deux approches idéales

- ☞ descendante (top-down)
 - *décomposition d'un problème en sous-problèmes*
 - *implantation ou réutilisation de composants qui résolvent les sous-problèmes*
- ☞ ascendante (bottom-up)
 - *création ou réutilisation de composants*
 - *composition des composants pour créer le nouveau système*

Une méthode « réaliste » utilise les deux approches



Décomposition

22

Comment arriver aux

- ☞ composants
- ☞ connecteurs
- ☞ la configuration du système

Quel est le niveau de granularité adéquat pour un composant ?

Quelles sont les contraintes imposées sur les composants par

- ☞ les besoins fonctionnels
- ☞ les besoins non-fonctionnels
- ☞ le dimensionnement du système
- ☞ les évolutions prévues
- ☞ l'environnement d'exécution
- ☞ les utilisateurs

Quelles sont les suppositions qu'un composant peut faire sur un autre ?



Décomposition

23

Comment les composants interagissent ?

Quels sont les connecteurs du système

Quel est le rôle des connecteurs ?

- ☞ médiation
- ☞ coordination
- ☞ communication

Quelle est la nature des connecteurs ?

- ☞ type d'interaction
- ☞ degré de concurrence
- ☞ degré d'échange d'information



Composition

24

Résumé et extension des problèmes de la réutilisation

- ☞ où sont localisés les
 - ➔ *composants existants*
 - ➔ *connecteurs existants*
 - ➔ *configurations existantes*
- ☞ comment détermine-t-on les éléments nécessaires ?
 - ➔ *Au moment du développement et au moment de la réutilisation*
- ☞ quel est le niveau de granularité adéquate pour un élément ?
- ☞ comment est effectué la composition des éléments hétérogènes ?
- ☞ comment est déterminé le fait que l'on ait le système nécessaire ?



Définitions de la notion de style

25

Les styles architecturaux sont **des patrons et des expressions** organisationnels et récurrents [Shaw & Garlan]

La compréhension établie et partagée de formes de conception communes est une **marque de maturité** dans le domaine de l'ingénierie du logiciel [Shaw & Garlan]

Le style architectural est une abstraction des caractéristiques de composition et d'interaction récurrentes d'un ensemble d'architectures [Taylor]

Les styles sont des **expressions de conception** qui permettent l'exploitation de patrons structurels et évolutifs adéquats et qui facilitent la réutilisation de composants, de connecteurs et de processus [Medvidovic]



Catégories de styles

26

Patrons et expressions

- ☞ structures d'organisation globale
 - *Ensemble de règles de bonne formation*
- ☞ indépendant du domaine d'application
 - *pipe & filtre*
 - *client/serveur*
 - *tableau noir (blackboard)*
 - *couche*

Modèles de référence

- ☞ configurations spécifiques pour certains domaines d'application
- ☞ peut être appliqués en dehors de leur domaine initial
- ☞ garant de certains attributs de qualité (maintenabilité, performance, fiabilité)

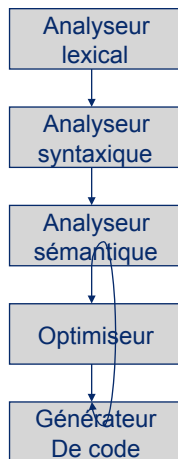


Exemple de style

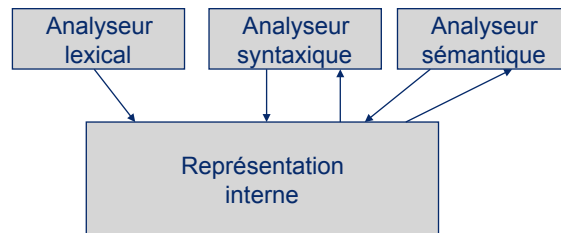
27

Un compilateur

Séquentiel



Parallèle



Séquentiel
+ Conception simple
+ L'architecture reflète le flot de contrôle
- Performance

Parallèle
+ Performance
+ Adaptabilité
- Synchronisation
- Coordination



INRIA centre de recherche LILLE - NORD EUROPE



Propriétés de base des styles

28

- ☞ Un vocabulaire d'éléments de conception
 - ☞ types de composants et de connecteurs
 - ☞ e.g. pipes, filtres, objets, serveurs
- ☞ Un ensemble de règles de configuration
 - ☞ des contraintes topologiques qui déterminent les compositions permises d'un ensemble d'éléments
 - ☞ e.g. un composant peut être connecté au plus à deux autres composants
- ☞ Une interprétation sémantique
 - ☞ les compositions d'éléments de conception ont un sens bien défini
- ☞ Les analyses possibles des systèmes construits selon un style
 - ☞ la génération de code est un type particulier d'analyse



INRIA centre de recherche LILLE - NORD EUROPE



Les avantages des styles

29

Réutilisation de la règles de conception

- ☞ des solutions bien comprises appliquées à de nouveaux problèmes

Réutilisation de code

- ☞ des implantations partagées des aspects invariants d'un style

La compréhension de l'organisation d'un système

- ☞ une expression telle que « client-serveur » comprend beaucoup d'information

L'interopérabilité

- ☞ gérée par la standardisation

- ☞ e.g. CORBA, EJB,...

Les analyses spécifiques aux styles utilisés

- ☞ permises par les espaces de conception contraints

Visualisation

- ☞ représentation du style en correspondance avec le modèle mental de l'ingénieur



Qualité dans les architectures logicielles

30

Une décision architecturale doit être conduite par une recherche de qualité

« Ce ne sont pas les fonctionnalités attendues qui conduisent les décisions architecturale mais la qualité recherchée »

Exemple de logiciel en couche -> recherche de maintenabilité

Quelques exemples de critères de qualité [ISO 9126]:

- ☞ capacité fonctionnelles
- ☞ portabilité
- ☞ fiabilité
- ☞ utilisabilité
- ☞ efficacité
- ☞ maintenabilité
- ☞ portabilité

Autre : testabilité, compréhension, flexibilité



III. Modèles & langages à Composants



Approche

Avoir une vision de l'architecture de l'application

- ☞ des composants logiciels utilisés et leur dépendance
- ☞ des modes de communication entre composants
- ☞ de la politique de répartition

Démarche

- ☞ le concepteur spécifie l'architecture
 - *par un langage déclaratif : langage d'interconnexion de module, langage de configuration, langage de définition d'architecture*
 - *analyse des propriétés de composition, de sûreté et vivacité*
- ☞ L'architecture est ensuite déployée
 - *Service système de déploiement*



Différents langages

33

Rôles des ADLs

- ☞ Supporter des abstractions
 - *spécifiques à l'application et à l'utilisateur*
- ☞ Fournir un langage et un environnement
 - *le langage permet des spécifications précises*
 - *l'environnement les rend (ré) utilisables*

Principaux ADLs

ACME ArchJava Darwin Aesop C2 Wright Rapide
Unicon Olan MetaH SADL Weaves xADL
AADL SafArchie ASAAM FRACTAL ArchWare UML 2.0



Langages de construction d'architectures à base de composants

34

Un ADL est un langage (formel ou semi-formel) qui fournit des dispositifs pour modéliser l'architecture conceptuelle d'un système logiciel, qu'on distingue de son implémentation

- ☞ Contrats d'interfaces et description des comportements
- ☞ Accompagnement par une démarche de construction ou d'évolution
- ☞ Prise en compte de la dynamique de l'architecture
- ☞ Abstraction du langage vis-à-vis d'une plate-forme d'exécution
- ☞ Déploiement sur une infrastructure physique

3 grandes familles

- ☞ Modèles à composants académiques : ArchJava, Fractal
- ☞ Langages de description d'architecture (ADLs) : Wright, Darwin
- ☞ Les standards : AADL, UML 2.0



III.1 Modèles à composants académiques



Définition d'un composant logiciel

Clemens Sziperski :

Un composant est une unité de composition qui spécifie par contractualisation ses interfaces et qui explicite ses dépendances de contexte uniquement. Un composant logiciel peut être déployé indépendamment et est sujet à composition par des entités tierces.

UML 2.0 :

Un composant est une partie modulaire d'un système qui encapsule son contenu et dont la manifestation est remplaçable dans son environnement. Un composant définit son comportement en termes d'interfaces fournies et requises. En tant que tel un composant sert comme un type, dont la conformité est définie par ses interfaces fournies et requises (incluant à la fois leur sémantique statique et dynamique)

Dans le domaine des architectures abstraites:

on se focalise sur la description abstraite du logiciel en vue d'une analyse des propriétés de qualités- on se limite aux phases d'analyse et de conception

Dans le domaine des plates-formes :

On s'intéresse au modèle d'exécution lié à l'intergiciel : implémentation, déploiement, exécution



Modèles à composants

37

☞ Modèles à composants

- *Composants = code applicatif*
- *Connecteurs = interaction, glue, adaptateur*
- *Points de liaisons : Ports*

☞ le mode de communication est « transparent »

☞ Technique de composition

- ☞ code d'adaptation et de glue via des connecteurs
- ☞ Séparation du code applicatif et des interactions

☞ Langage de composition : un ADL est un langage simple de composition



Composant et Composition

38

Composant = partie servant à composer un tout
Souvent boîte noire : spec. externe + code

Composition = action de composer un tout par l'agencement de plusieurs parties

différents mécanismes de composition

- ☞ **Structurelle** : Etablissement et contrôle des liaisons entre composants
 - *Interaction (services fournis/requis)*
 - *Implantation (décomposition hiérarchique)*
- ☞ **Comportementale** : Composition d'automates
- ☞ **Contractuelle (QoS)** : Composition de contrats comportant des obligations et/ou des hypothèses

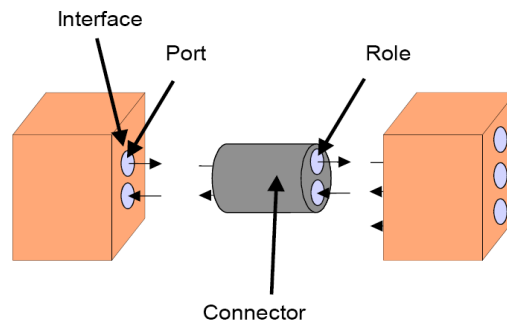


Modèle à composant dans une architecture

39

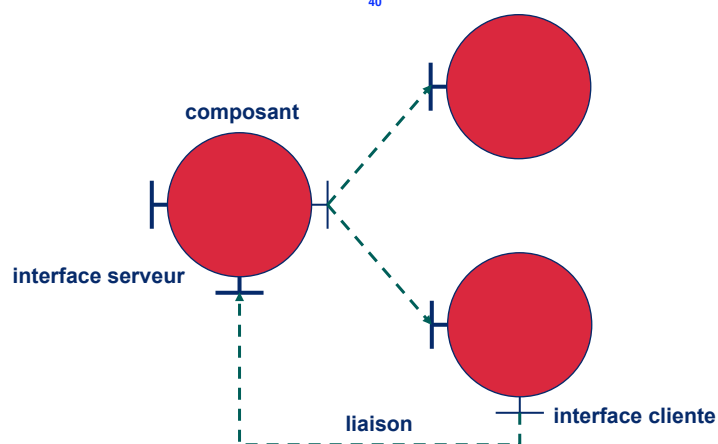
Port : points d'interfaces abstraits (événements, messages)- spécifie les flux entrants et sortants d'un composant

Connecteur : composant dédié à la communication - les connecteurs sont liés à des ports via des rôles



Modèle à composants Vue abstraite

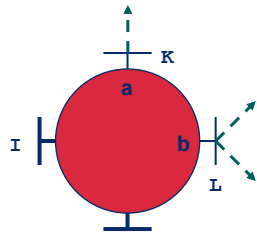
40



Modèle à composants

Typage

41



```
interface J {
    void m ();
    // ...
}
```

Type =

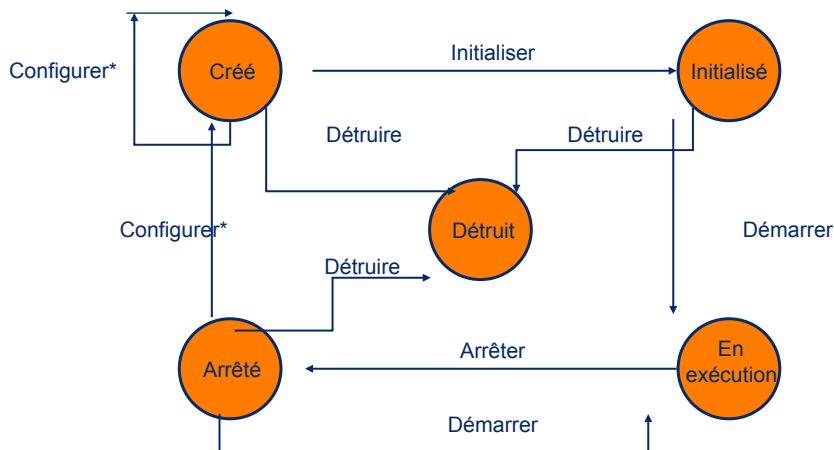
- types des interfaces serveur =
 - I
 - J
- types des interfaces clientes =
 - a → { K, obligatoire, 1 → 1 }
 - b → { L, optionelle, 1 → n }



Modèle à composants

Cycle de vie

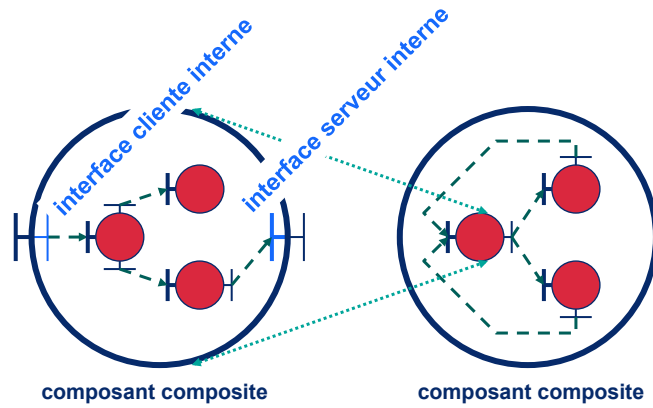
42



Modèle à composants

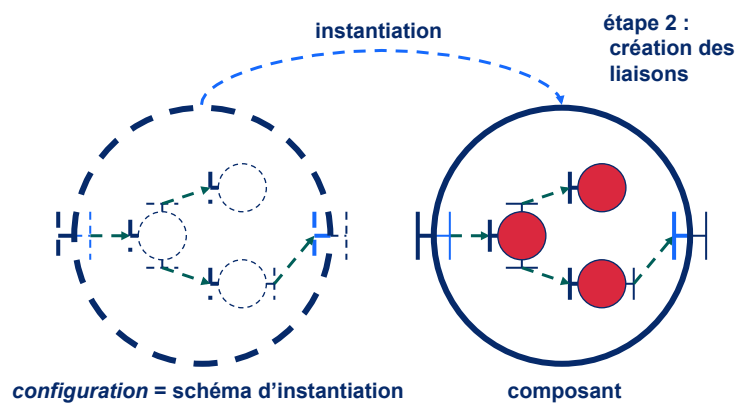
Composites

43



Instantiation de composants

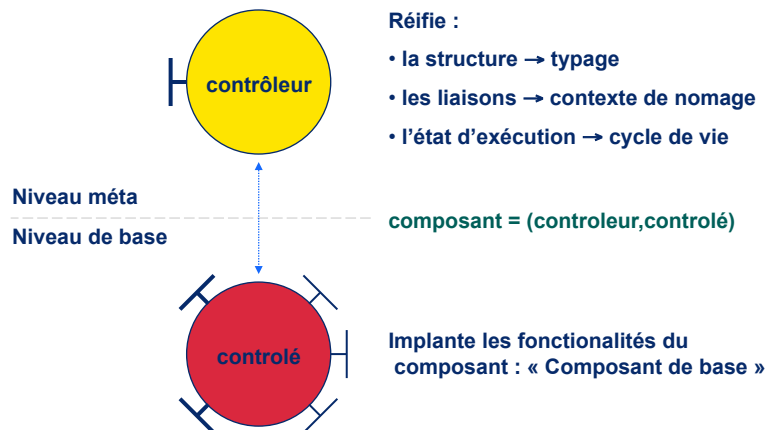
44



Modèle à composants

Niveaux d'exécution

45



ArchJava

46

Univ. Washington [Aldrich 2002]

Langage de description d'architecture à la frontière entre

- un modèle à composants
- un ADL

Objectifs :

- améliorer la compréhension des programmes
- garantir l'architecture de l'application
- assurer l'évolutivité des applications
- encourager les développeurs à se servir des concepts de composants, connecteurs, configurations



ArchJava

47

Le composant ArchJava

- unité de traitement + interface décrivant les services fournis et requis
- composant primitif et composant composite
- un composite peut contenir du code logiciel

Le connecteur ArchJava

- communication à l'aide de l'appel de méthode
- garantie des com. entre composants par respect des connexions définies dans l'architecture
- vérification par compilation
- définition de classes de connexions appelées connecteurs complexes

La configuration ArchJava

- composants composites = unité de configuration



Quelques propriétés d'ArchJava

48

Déploiement

- pas de modèle de déploiement - pas de localisation des éléments
- à faire manuellement

Dynamique

- création de composants et de liaisons dans les composites
- toute nouvelle connexion à l'exécution doit être déclarée afin de garantir la capacité d'analyse à la compilation
- prise en charge limitée
 - pas de possibilité de destruction
 - pas de possibilité de déplacement d'instance ds un composite

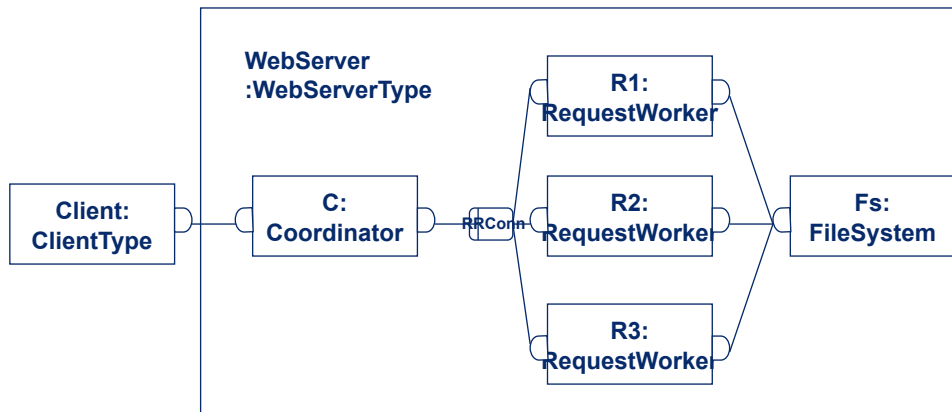
Comportement

- pas de possibilité de description du comportement



Exemple

49



Un peu de description textuelle

50

```
Public component class WebServer
private final Coordinator c= new coordinator();
private final RequestWorker R1 = new RequestWorker(« R1 »);
private final RequestWorker R2 = new RequestWorker(« R2 »);
private final RequestWorker R3 = new RequestWorker(« R3 »);
private final FileSystem fs = new FileSystem(new File (« /home/test »));

// déclaration des liaisons
connect c.serviceHttp, rserviceHttp;
connect R1.handleRequest, c.handleRequest with RoundRobinConnector.getConnector (connection);
connect R2.handleRequest, c.handleRequest with RoundRobinConnector.getConnector (connection);
connect R3.handleRequest, c.handleRequest with RoundRobinConnector.getConnector (connection);
connect R1.AccessFS, fs.AccessFS;
connect R2.AccessFS, fs.AccessFS;
connect R3.AccessFS, fs.AccessFS;

Public port serviceHttp{
provides String get (URL url) {...}
}

Public port rserviceHttp{
requires String get (URL url) {...}
}
}
```



Un peu de description textuelle

51

```
public component class Coordinator{
  public port serviceHttp{
    provides String get (URL url){...}
  }
  Public port handleRequest{
    requires String handleRequest(URL url);
  }
}

public component class RequestWorker {
  private String _name;
  public RequestWorker(String name){this._name=name;}

  Public port AccessFS {
    requires boolean open (File f);
    requires String read (File f);
    requires void close (File f);
  }
}
```



Un peu de description textuelle

52

```
Public component FileSystem {
  public port AccessFS{
    provides boolean open (File f){...}
    provides String read (File f){...}
    provides void close (File f){...}
  }
}

Public class RoundRobinConnector extends Connector {

  private port otherPort;
  private ArrayList ports = new ArrayList();
  private static RoundRobinConnector instance = null;

  Public static RoundRobinConnector getConnector (archjava.reflect.Connection c){
  ...
  }
}
```



Conclusion avec ArchJava

53

Approche orientée langage de programmation
Compilateur utilisé sur des applications de taille importante

Avantages :

- garantie de l'intégrité de l'architecture pendant la phase de développement
- les communications respectent la structure de l'application
- bonne traçabilité de la structure
- compilateur éprouvé + Java -> bonne attractivité

Inconvénients

- reste proche de l'implantation - travaux avec ACME pour couche d'abstraction supplémentaire
- spécification de la structure uniquement
- mélange architecture et objet -> complexité du code



Conclusion sur les modèles à composants

54

Très nombreux modèles

Pouvoirs d'expression très variés

- pb d'assemblage (structurel, comportemental, architectural, contractuel)
- pb fonctionnel
- pb techniques
- évolution/adaptabilité
- qualités
- ...



III.1 Les ADLs Classiques



Wright

Wright [Allen 97]:

56

Description de comportement des composants
spécification de l'architecture et de ses éléments
pas de générateur de code, ni de plate-forme de simulation

Composant

Unité de traitement abstraite localisée et indépendante
Instance de composant/ sémantique = processus
- Interface = ens. de Ports / spécification formelle CSP
- Partie calcul = Comportement du composant

Connecteur

Interaction entre une collection de composants
Réutilisable
deux parties : rôle (CSP) et glue(interaction entre les rôles)

Configuration

architecture du système
déclaration des composants et des connecteurs,
déclaration des instances des composants et connecteurs,
description des liens entre les instances de composants



Wright

57

Orienté vérification formelle
uniquement la description des composants et des interactions

Dynamique
description à l'aide d'un « configurateur »
langage ad-hoc pour définir les reconfigurations

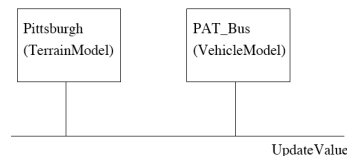
Comportement
utilisation de CSP
description du comportement du processus
pas de variable commune - communication par message - RDV
exécution en parallèle

Analyse de propriétés
propriétés de sûreté : rien de mauvais n'arrive
propriété de vivacité : quelque chose de bon peut toujours arriver



Exemple

58



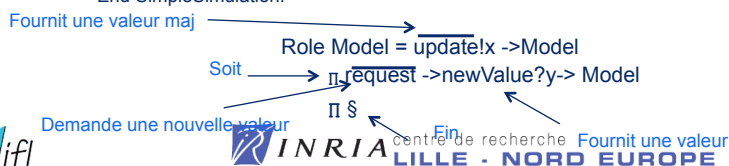
```

Configuration SimpleSimulation
  Component TerrainModel(map : Function)
    Port ProvideMap = [Interaction Protocol]
    Computation = [provide terrain data]
  Component = VehicleModel
    Port Environment = [Interaction Protocol]
    Computation = [compute vehicle movement]
  Connector UpdateValues(nsims : 1..)
    Role Model 1::nsims = [Interaction Protocol]
    Glue = [Data travels from one Model to another]

Instances
  Pittsburgh : TerrainModel([map of Pittsburgh])
  PAT Bus : VehicleModel
  C : UpdateValues(2)

Attachments
  Pittsburgh.ProvideMap, PAT Bus.Environment as C.Model

End SimpleSimulation.
  
```



Une autre forme avec style

59

Style Simulation

```

Interface Type SimInterface = [Interaction of one simulation]
Connector UpdateValues(nsims : 1..)
  Role Model 1..nsims = SimInterface
  Glue = [Data travels from one Model to another]
Constraints
  ∃ C : Connectors | {C} = Connectors
  ^ Type(C) = UpdateValues
  
```

End Style.

Configuration SimpleSimulation2

Style Simulation

```

Component TerrainModel(map : Function)
  Port ProvideMap = SimInterface
  Computation = [provide terrain data]
Component = VehicleModel
  Port Environment = SimInterface
  Computation = [compute vehicle movement]
  
```

Instances

```

Pittsburgh : TerrainModel([map of Pittsburgh])
PAT Bus : VehicleModel
C : UpdateValues(2)
  
```

Attachments

```

Pittsburgh.ProvideMap, PAT Bus.Environment as C.Model
End SimpleSimulation2
  
```



centre de recherche
LILLE - NORD EUROPE



Exemple : Architecture Client/serveur avec Wright (CMU)

Détenteur du compte bancaire / Client

Compte bancaire/ Serveur



COMPONENT Client

PORT Appel-debiter = Requete → reponse → Appel-debiter [] \$

COMPUTATION = TraitementInterne → Appel-debiter.Requete → Appel-debiter.reponse → COMPUTATION [] \$

COMPONENT Compte

PORT Debit-compte = requete → reponse → Debit-compte [] \$

COMPUTATION = TraitementInterne → Debit-compte.requete → Debit-compte.reponse → COMPUTATION [] \$

CONNECTOR Connecteur-RPC

ROLE appelant = Requete → reponse → appelant [] \$

ROLE appele = requete → reponse → appele [X] \$

GLUE = appelant.requete → appele.requete → GLUE

[X] appelant.reponse → appele.reponse → GLUE

[X] \$

CONFIGURATION Application-bancaire

COMPONENT Client

COMPONENT Compte

CONNECTOR Connecteur-Rpc

INSTANCE

client1 :Client;

compte1 :Compte;

appel-rpc : Connecteur-RPC ;

ATTACHMENTS

client1.Appel-debiter as appel-rpc.appelant;

compte1.Debiter-compte as appel-rpc.appelle;

End Application-bancaire



centre de recherche
LILLE - NORD EUROPE



Conclusion sur Wright

61

Avantages

- langage formel CSP pour les composants et des connecteurs
- description des comportements
- analyse de propriétés
- communication sans ambiguïté entre les acteurs

Inconvénients

éloigné des habitudes du développeur
description pauvre des contrats (syntaxique et sémantique)
pas d'environnement d'exécution - pas de génération de code
ne supporte pas la répartition

Travail à faire

Décrire l'exemple web serveur en Wright

http://www.cs.cmu.edu/afs/cs/project/able/www/wright/wright_bib.html



Darwin

62

Imperial College [Magee 95]

- Modèle de composants pour la construction d'applications réparties
- Spécification de la dynamique d'une application

Composant

instance définie par une interface fournie et requise
sémantique associée : processus
chaque composant est un processus

Connecteur

pas explicit
chaque interaction est représentée par un lien entre serv. fourni et serv. requis
l'objet port permet la définition des objets de communication

Configuration

composant primitif - composant composite



Darwin

63

Support d'exécution spécifique

- ☞ Regis : C++ pour construire et exécuter des archi. Darwin

Déploiement

- ☞ en accord avec Regis
- ☞ association site et composant

Description de la dynamique

- ☞ schéma d'instanciation des composants - création dynamique des composants
- ☞ instanciation paresseuse : pré-déclaration des instances à créer lors du premier appel
- ☞ instanciation dynamique : création d'instance en lui fournissant les paramètres d'initialisation - se fait au sein d'un composite

Description de comportement

- ☞ Extension Tracta- description des comportements : FSP/LTS
- ☞ propriétés de sûreté et de vivacité



Un exemple : Darwin

```
Component Client (int nbAnuaire) {
  require lookupAnnuaire[nbAnuaire]<port,string,string>;
}
Component Annuaire(int pays){
  provide lookup <port,string,string>
}
Component ClientFactory{
  require createClient <component>;
}
```

```
Component Application {
  const int nbMaxAnnuaire=2
  inst clientFact : ClientFactory; // créateur de clients
  array Annuaire[0..nbMaxAnnuaire-1]:Annuaire // ensemble d'annuaires
  bind clientFact.createClient-dyn Client(nbMaxannuaire);
  //création dynamique de clients
  forall k:[0..nbMaxAnnuaire-1] //itérateur-crée les annuaires et
    // les interconnexions entre tout client et les annuaires
    inst Annuaire[i] : Annuaire(i)
    bind Client.lookupAnnuaire[k] - Annuaire[k].lookup;
```



Darwin : Evaluation

65

Intégration

- rien de fait

Structure de l'application

- Notion de groupe d'instances limité
- Interconnexions sont des liaisons
 - *Protocole de communication : utilisation de mécanisme de communication est à la charge du programmeur*

Dynamique

- Prise en compte
- Description des contrats de comportement

Répartition

- Configurable



Bilan sur les ADLs

66

Avantages

- Conçus spécifiquement pour les architectures - DSLs
- Plan global d'application : modèle pivot de modélisation
- Séparation claire entre composants et interactions

Inconvénients

- manque d'effort commun de la communauté pour un langage commun
- Reste un travail académique - difficile à prendre en main
- Passage à l'échelle ?
- Peu de description de la dynamique



En résumé

67

A qui servent-ils ?

- ☞ Description de l'architecture intégrant
 - *La description hiérarchique des composants logiciels*
 - *Spécification externalisée de la communication*
 - *Spécification d'une partie du comportement dynamique*
 - instanciation/suppression de composants
 - de la communication
- ☞ Modèle d'intégration et de description et logiciels existants
 - *Traitement de l'hétérogénéité des types de logiciels (exécutables, code interprété, bibliothèques,...)*
 - *De la plate-forme d'exécution*
 - *Du modèle d'exécution propre aux composants*
- ☞ Pour le déploiement
 - *En lien avec la définition de l'architecture*
 - *Étude de structures d'exécution adéquates*



68

III.3 Les Standards



AADL

69

Dérivé de MetaH[Vestal 1998]

Dédié aux systèmes avioniques (Darpa et US Army)

Effort de standardisation à partir de 2001 : AADL

AADL : Architecture Analysis & Design Language

Par la SAE : Society of Automotive Engineers

Très bon accueil des industriels

Besoin spécifique des systèmes embarqués temps-réel

Spécification de la QoS : exigences temporelles et spatiales,
fautes, erreurs, sûreté, certification...



AADL

70

Langage textuel & graphique

Composant:

- description du type de composant : interface fonctionnelle
- description de l'implantation : contenu du composant
- catégorie pré-définie : mémoire, bus, périphérique, processeur, donnée, ...
- communication par l'intermédiaire de ports : data, event, data event

Connecteur

- définition de connexions entre ports in et out
- possibilité d'ajout de propriétés ex: délai de transmission)

Configuration

- modèle hiérarchique



AADL

71

Propriétés

- caractérisation du composant par un couple attribut-valeur
- ex: processus léger : période, échéance, durée d'exécution

Annexes

- utilisation de déclaration écrites dans d'autres sous-langages
- précision facultative



AADL

72

Déploiement

- prise en compte des éléments matériels
- maximum d'infos sur l'infrastructure

Dynamique

- mécanisme de gestion de la reconfiguration
- notion de mode : différentes configurations admissibles d'un système - évolutions possibles

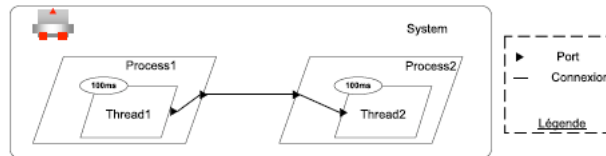
Comportement

- pas complètement défini - diagramme de flot
- pas de description des interactions



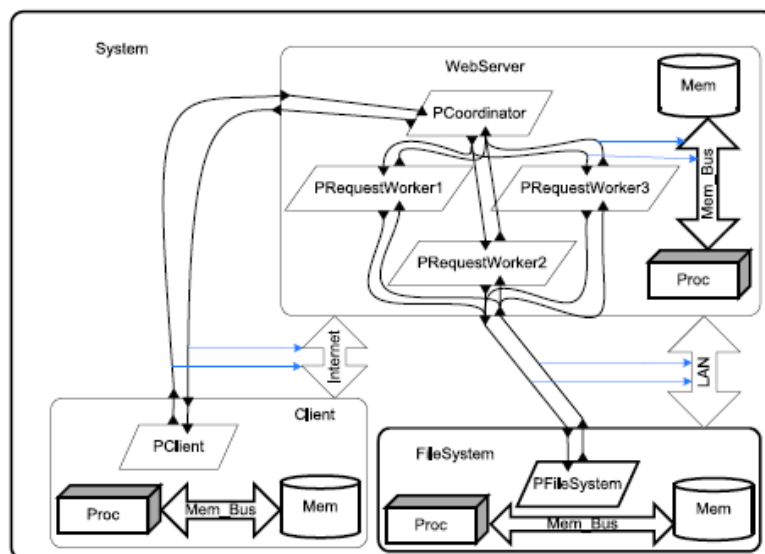
Exemple AADL

73



Un autre exemple

74



Conclusion AADL

75

Avantages

- monde de l'industrie du transport
- prise en compte de sémantiques différentes au niveau du composant / description des éléments matériels
- notion de propriétés et d'annexes : ajout à la demande

Inconvénients

- pas de comportement pour spécifier les relations entre E/S d'un composant
- pas de sémantique claire de la composition
 - ☞ outils ad-hoc pour les propriétés, mais pas de sémantique définie au niveau de la norme pour pb de composition
- très orienté mise en œuvre / pas utilisable tout le long de la conception



UML 2.0

76

Introduction d'un diagramme d'architecture

- diagramme de structure composite
- comble les lacunes de la 1ere version

Composant

- entité modulaire et réutilisable avec interfaces R/F
- interface : définition du type du composant
- interface attachée à un port qui est point d'interaction du composant
- composant basique : étend la notion de classe
- composant empaqueté : entité composée d'autres éléments

Connecteur

- décrit les interactions
- relie les ports d'un composant à un autre composant
- connecteur de délégation pour connecter un port externe au port d'un sous-composant interne

Configuration

- Notion de partie (Part) : description de la partie interne d'une classe
- structure interne d'un composant : ensemble de parties
- modèle hiérarchique



UML 2.0

77

Modèle partiellement défini

- la sémantique de certaines constructions sont floues
 - attachement de plusieurs connecteurs à un même port : pas de description comportementale
- point de variation sémantique
- rend UML 2.0 difficilement utilisable tel quel pour décrire une architecture



UML 2.0

78

Déploiement

- ☞ prise en compte du déploiement sur une infrastructure physique répartie
- ☞ ressources = nœuds interconnectés entre eux
- ☞ pas de processus de déploiement, ni information minimale

Dynamique

- ☞ diagramme de séquences : création et destruction d'instances
- ☞ aucune opération liée à la reconfiguration ou à la migration

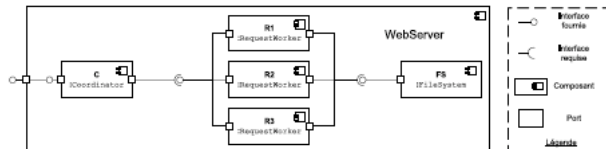
Spécification du comportement

- ☞ diagramme de séquences : définition des interactions entre deux entités d'application
- ☞ MSC (Message Sequence Charts) - outils de validation



Exemple UML 2.0

79



Conclusion UML 2.0

80

Convergence avec les ADLs et le modèle de composant UML 2.0

Mais

- ☞ relations entre les concepts mal définies
- ☞ signification des constructions trop imprécises
- ☞ exemple
 - le comportement d'un composant peut être modélisé par une machine à états
 - un port peut se voir attaché un comportement
 - un composite peut se voir attacher une collaboration
 - rien n'est assuré de la cohérence de ces 3 informations
- ☞ nécessité de redéfinir ou de préciser la sémantique avec des profils UML ou des DSL.



IV. Un modèle d'évolution d'architecture logicielle

Sommaire

Introduction
Que faut-il adapter ?
Quand faut-il adapter ?
Mise en œuvre de l'adaptation
Où gère-t-on l'adaptation ?
Qui gère l'adaptation ?
Comment adapter ?
Techniques pour l'adaptation
Un modèle d'évolution d'architecture logicielle
Conclusion

Introduction

83

Contexte

- ↳ Environnement hétérogène et ouvert
- ↳ Environnement ubiquitaire - **variation spatiale**
- ↳ Evolution des besoins et des technologies - **variation temporelle**

Définitions

Adapter au sens général : *rendre (un dispositif, des mesures) apte à assurer ses fonctions dans des conditions particulières ou nouvelles*

Adapter un système logiciel : *processus de modification nécessaire pour permettre un fonctionnement adéquat dans un contexte donné*

Adaptabilité : *qualité de ce qui peut être adapté, de ce qui peut s'adapter*



Introduction

84

Modularité et composition

- ↳ **Modularité** (pour ne modifier que ce qui est nécessaire)
 - ↳ Granularité plus ou moins fine
 - ↳ Exemples de modules : bibliothèque de code, une classe, un composant, un aspect
- ↳ **Composition** d'éléments logiciels modifiables (et réutilisables)
 - ↳ Notion de plug and play
 - ↳ Connecteurs (ADL)
 - ↳ Tissages de modules (Aspect)
- ↳ **Définition de variations structurelles, comportementale ou de QoS**
 - ↳ Au niveau des modules
 - ↳ Au niveau des interconnexions



Introduction

85

Adaptation aux différents niveaux du cycle de vie

- ↳ Analyse des besoins
- ↳ Conception
- ↳ Élaboration du code
- ↳ Déploiement
- ↳ Exécution



Que faut-il adapter ?

86

Trois éléments à adapter

- Le module logiciel
- La liaison
- L'architecture



Que faut-il adapter ?

87

Le module logiciel



Exemples

- ☞ Composant
- ☞ Aspect
- ☞ Service web
- ☞ Librairie

La description

- ☞ Son interface
- ☞ Son comportement
- ☞ Sa sémantique
- ☞ Sa qualité de service

Application d'un processus de modification

- Du simple paramétrage au changement complet



Que faut-il adapter ?

88

La liaison



Exemples

- ☞ Lien d'héritage
- ☞ Bus logiciel
- ☞ Connexion entre composants logiciels

La description

- ☞ Son comportement
- ☞ Sa sémantique
- ☞ Sa qualité de service

Application d'un processus de modification

- ☞ Changement sémantique du lien
- ☞ Adaptateur



Que faut-il adapter ?

89

L'architecture



Exemples

- ↳ Organisation des modules

La description

- ↳ Assemblage de modules
- ↳ Notion de patrons d'architectures

Application d'un processus de modification

- ↳ Ajout, retrait, remplacement de certaines boîtes
- ↳ Reconfiguration de liaisons entre boîtes



Que faut-il adapter ?

90

Adaptation fonctionnelle

- Propriété applicative - besoin métier
- Ajout/modification/retrait de services fonctionnels

Adaptation extra-fonctionnelle

- Propriétés du système (sécurité, transaction, persistance)
- Ajout/modification/retrait de contraintes sur le comportement des services fonctionnels



Quand ?

91

Avant l'exécution -- statique

↳ Adaptation au niveau conceptuel

- ↳ Concevoir des logiciels qui supportent l'adaptation
- ↳ Prévoir des possibilités d'adaptabilité et faire des choix en conséquence
 - Représentation de structures permettant l'expression de l'adaptabilité
 - ADL, Aspects, BPMN/Web services

↳ Adaptation du code

- ↳ 2 moments possibles : à la compilation et à l'édition des liens

Au lancement -- statique

↳ Au moment du déploiement

- ↳ Topologie
- ↳ Etat de l'environnement d'exécution

Pendant l'exécution -- dynamique

↳ Modification du sujet d'adaptation

↳ Reconfiguration dynamique

↳ Exemple : installation à la volée d'un composant, modification du bytecode



Mise en œuvre de l'adaptation

92

Etapes de l'adaptation

↳ Déclenchement

- ↳ Moments : à la compilation, au déploiement, à l'exécution
- ↳ Notification d'un événement
- ↳ Différents déclencheurs : sonde logicielle/matérielle, utilisateur

↳ Décision

- ↳ Réagir au changement détecté
- ↳ Définir une stratégie d'adaptation
- ↳ Déterminer les différentes actions à effectuer
- ↳ Programmeur, déployeur

↳ Réalisation

- ↳ Moyens à mettre en œuvre pour réaliser la décision définie à l'étape précédente
- ↳ Déployeur, intergiciel, application

Si ces étapes se passent à l'exécution et sont automatisées, on parle d'**autonomic computing**



Où gère-t-on l'adaptation ?

93

Gérer l'adaptation à tous les niveaux ?

- ☞ Applicatif, intergiciel, système/réseau/matériel

Sujets à discussion

- ☞ Adapter l'intergiciel pour prendre en compte les besoins de l'application ?
- ☞ Adapter l'intergiciel pour prendre en compte les contraintes matériels/systèmes ?
- ☞ Adapter l'application aux services offerts par l'intergiciel ?
- ☞ Adapter les applications par l'intergiciel ?

Rôle et adaptation des intergiciels

- ☞ Cacher aux applications la répartition des ressources et l'hétérogénéité des plates-formes
- ☞ Surveillance des ressources, prise de décision, mise en œuvre de l'adaptation et information des changements aux applications



Qui gère l'adaptation ?

94

Adaptation manuelle

- ☞ Adaptation statique
- ☞ Outillage logiciel : pas d'intrusion nécessaire
- ☞ Acteurs du logiciel : concepteurs, architectes, développeurs, intégrateurs

Adaptation automatique

- ☞ Génération automatique des adaptations
- ☞ Outillage logiciel : détection, plan d'action, mise en place
- ☞ Acteurs du logiciel : développeurs spécialisés dans l'adaptation



Comment adapter ?

95

Caractérisation du sujet d'adaptation : module, liaison, architecture



Ce que l'on connaît du sujet à adapter

- ↳ L'interface, y compris le comportement à l'interface
- ↳ Le comportement (code) du module ou de la liaison
- ↳ L'état du module ou de la liaison
- ↳ La qualité de service

Critères de choix

- ↳ Modification de la sémantique ?
- ↳ Préservation de la sémantique ?



Comment adapter ?

96

Adaptation du module

- ↳ Changement de l'interface
 - ↳ *Sous-typage, héritage, autre*
- ↳ Paramétrisation
 - ↳ *Configuration du module / boîte noire*
- ↳ Transformation du code
 - ↳ *Modification du contenu du module / boîte blanche*
 - Spécialisation de programme
 - Tissage d'aspects



Comment adapter ?

97

Adaptation de la liaison

☞ Interposition

- Association squelette + talon + protocole de communication + réseau
- Interception - ajout d'un module pour gérer l'interception
- Encapsuleurs - conteneur

☞ Délégation

- Délégation du travail d'une entité à une autre entité
- Génération d'une autre liaison vue comme une extension de la liaison initiale
- Explicite si celle-ci est introduite par le concepteur
- Implicite si généré par des techniques de compilation ou de chargement
Transformation du code



Comment adapter ?

98

Adaptation de l'architecture

☞ Procédure de configuration/reconfiguration

- Composition/recomposition des modules
- Retrait/remplacement/paramétrage des modules et liaisons
- Ajout de nouveaux modules/liaisons

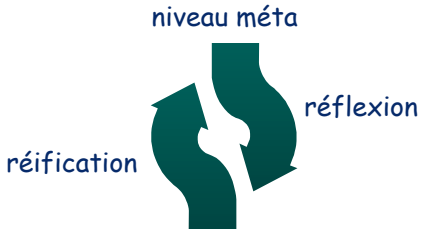


Techniques pour l'adaptation

99

Réflexivité : la capacité d'un système "à raisonner et à agir sur lui-même" [Smith 82][Maes 87]

- niveau de base
 - ⇒ l'application "standard"
- niveau méta
 - ⇒ réifie la base
 - ⇒ l'instrospecte
 - ⇒ la modifie



Catégories de réflexivité

- ⇒ à la compilation, à l'exécution, au chargement (Java)
- ⇒ structurelle (ex : API java.lang.reflect)
- ⇒ comportementale

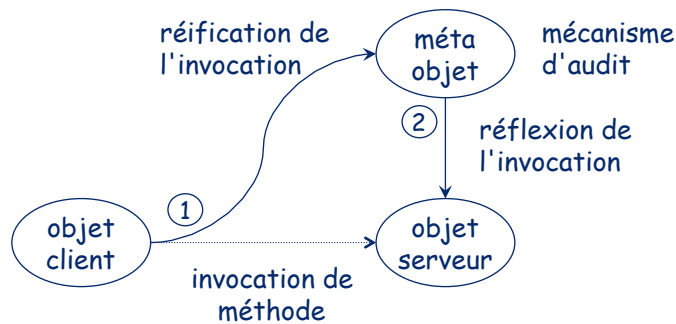
Performance vs flexibilité



Techniques pour l'adaptation

100

Exemple Réflexivité comportementale à l'exécution



Lien 1-1 (ou 1-n) entre l'objet et son méta
 ⇒ permet d'étendre le comportement du serveur sans le modifier



Techniques pour l'adaptation

101

Aspects et tissage

Point de jonction (join point)

- ☞ élément du prog. de base
- classe, méthode, exception**
- interface, donnée, structure de ctrl
- argument méthode, ...

Méthode d'aspect (advice)

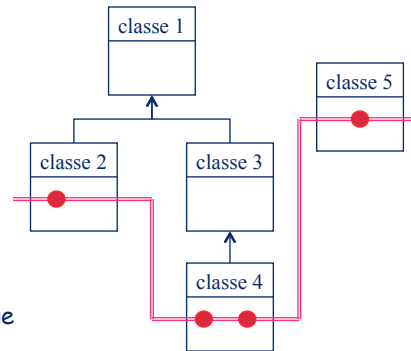
- ☞ traitement additionnel

Coupe transversale (pointcut)

- ☞ ens de pts de jonction ayant un lien logique entre eux

Aspect

- ☞ coupe + méthode d'aspect



AspectJ

102

Exemple AspectJ (<http://www.eclipse.org/aspectj/>)

aspectj.org

Le langage de référence de l'AOP

```
aspect Log {  
  pointcut traceMethode():  
    call( void Figure.set*(int) ) ||  
    execution( ! public static * com.xerox.*.* *(..) );  
  before(): traceMethode()  
  { System.out.println(".."); }  
}
```

définition d'une coupe
= ens de pts de jonction

définition d'une
"méthode" (advice)
associée à cette coupe



Techniques pour l'adaptation

103

Architectures logicielles

- ☞ Abstraction d'un système logiciel
- ☞ Distinction entre modules et interactions
- ☞ Accessibilité par les interfaces - port et rôles
- ☞ Adaptation par configuration/reconfiguration

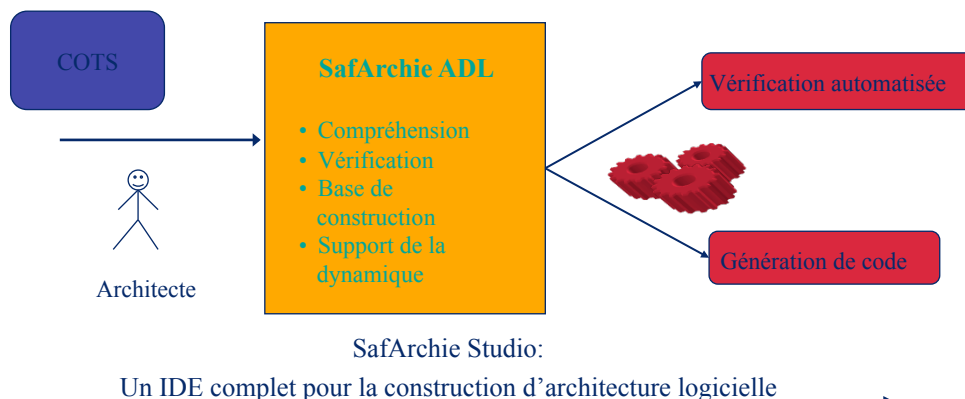
Limites

- ➔ *Un seul point de vue par un MM figé*
- ➔ *UML vs ADL extensible*



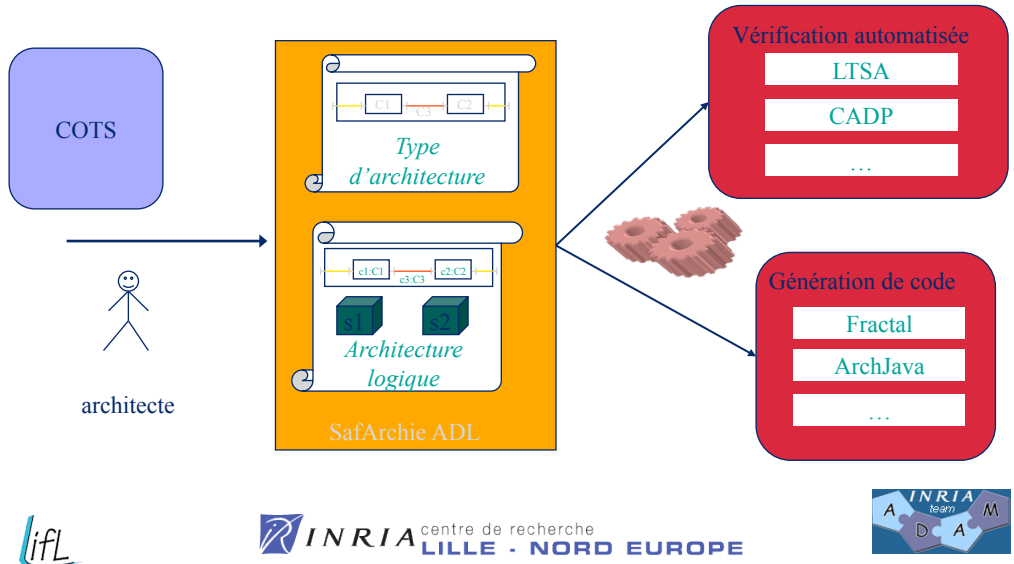
Un modèle d'évolution d'architecture logicielle

104



Vue globale du projet

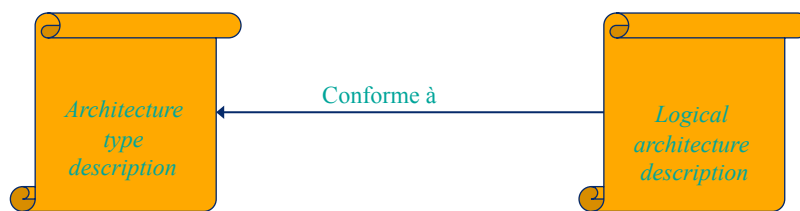
105



Le modèle SafArchie

106

Deux niveaux de description

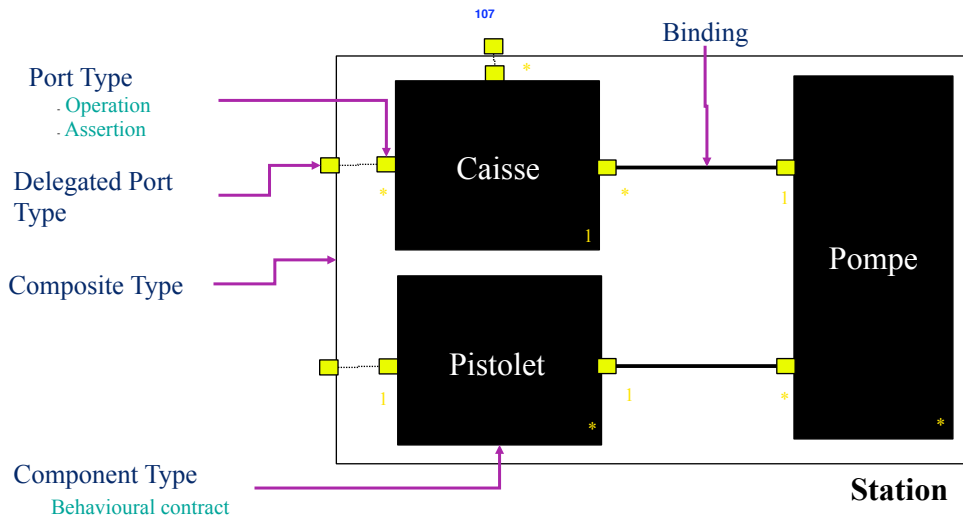


- Respects des invariants
- Vérifications

- Dynamique
- Etat du système
- Déploiement



Type d'architecture



Modèle d'architecture logicielle : Exemple de description textuelle

108

```

component CashRegister {
  Port Payment {
    require void start();
    provide void stop(int price);
  }
  Port Auth {
    require boolean askauth(string clientID);
    require boolean sale(string clientID, int price);
  }
  Port UserInterface {
    provide void putcard(Card c);
    provide void entercode(int code);
    require void givecard(Card c);
    require void givereceipt(Ticket t);
  }
}

component Station {
  //Declaration des composants du composite
  include Cash_Register, Gun, Pump;

  // Declaration de l'assemblage
  bind CashRegister.Payment Pump.PaymentFunction;
  bind Pump.ProvideGas Gun.GetGas;

  // Declaration des ports du composite
  Port ProvideGas redirect Gun.ProvideGas;
  Port UserInterface redirect CashRegister.UserInterface;
  Port Auth redirect Cash_Register.Auth;
}
    
```



Les contrats

109

Les assertions

- ☞ Expression de logique du premier ordre sur les types de paramètres des opérations

```
Require void sale(int price)
  @[PRE:self.parameter.price > 5,
  @POST : true];@*/
```

Le contrat de comportement

- ☞ Primitif : Traces finies indéterministes des messages que le composant peut échanger avec son environnement

```
GUN = (guit.getgaz -> pgt.providegaz -> GUN) +
  {guit.ClientStation2, pgt.PumpGun}.
```

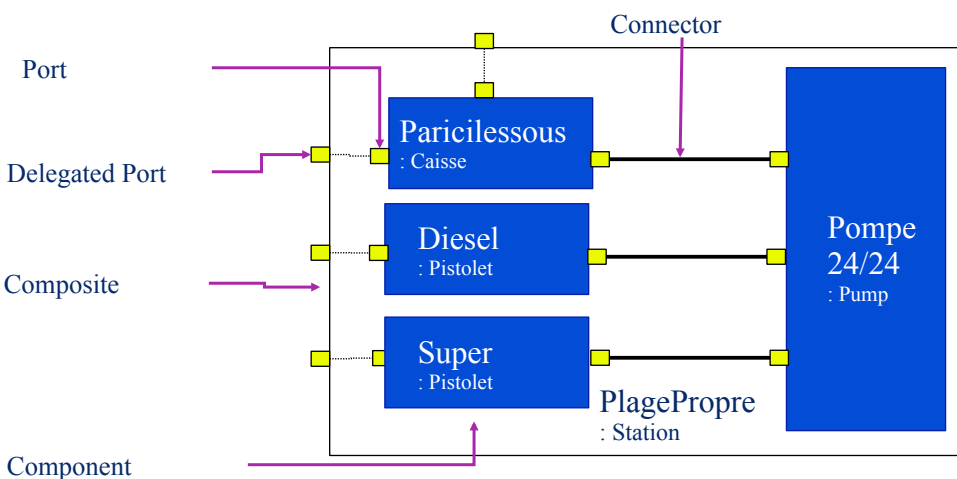
- ☞ Composite : Description de l'assemblage et des communications possibles vers l'extérieur du composite

```
||STATION = (CASHREGISTER || PUMP ||GUN) @ {cruil.ClientStation1,
  guit.ClientStation2, aut.BankStation}.
```



Architecture logique

110



Vérification automatisée

1/3

111

Compatibilité structurelle entre types de port reliés

```
<PortType PortID="port5" Name="PumpCR">  
<Operation Name="stop" Sens="require" Return="Void">  
  <Parameters Name="price" Type="Integer"></Parameters>  
</Operation>  
....  
</PortType>  
....  
  
<PortType PortID="port4" Name="CRPump">  
<Operation Name="stop" Sens="provide" Return="Void">  
  <Parameters Name="price" Type="Integer"></Parameters>  
</Operation>  
....  
</PortType>
```

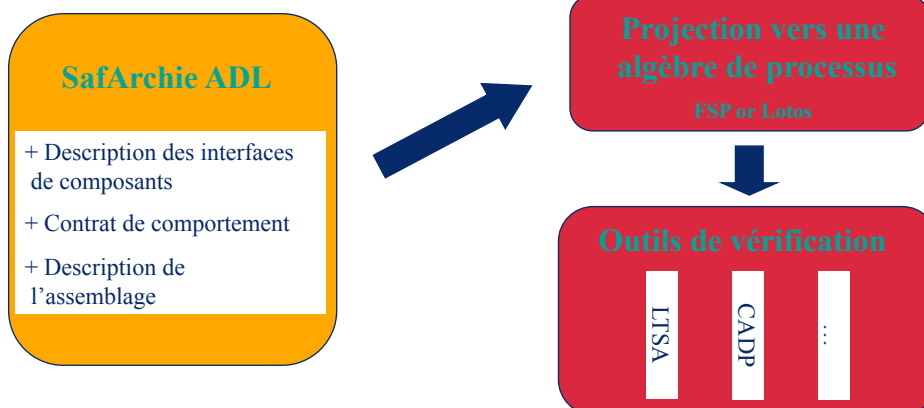


Vérification automatisée

2/3

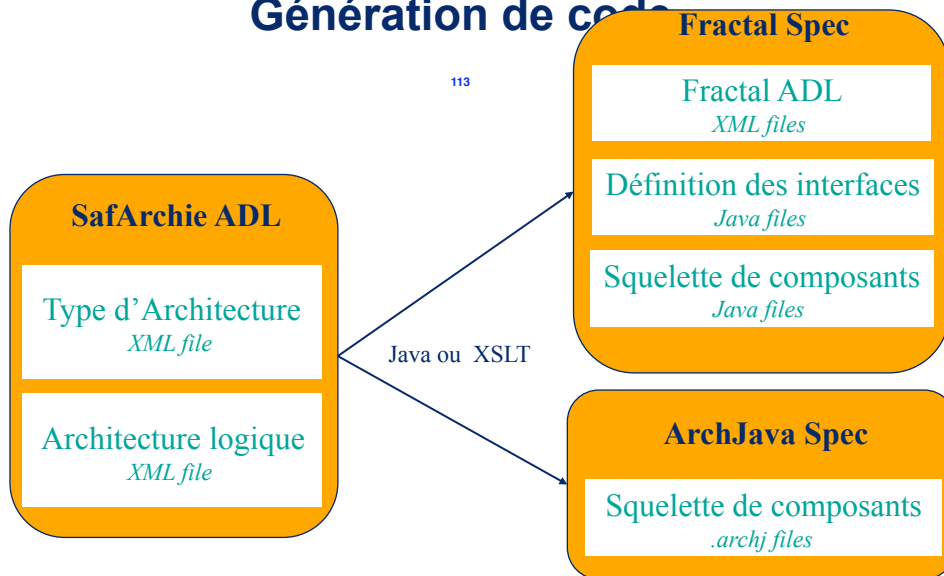
112

Synchronisation de types de composant liés



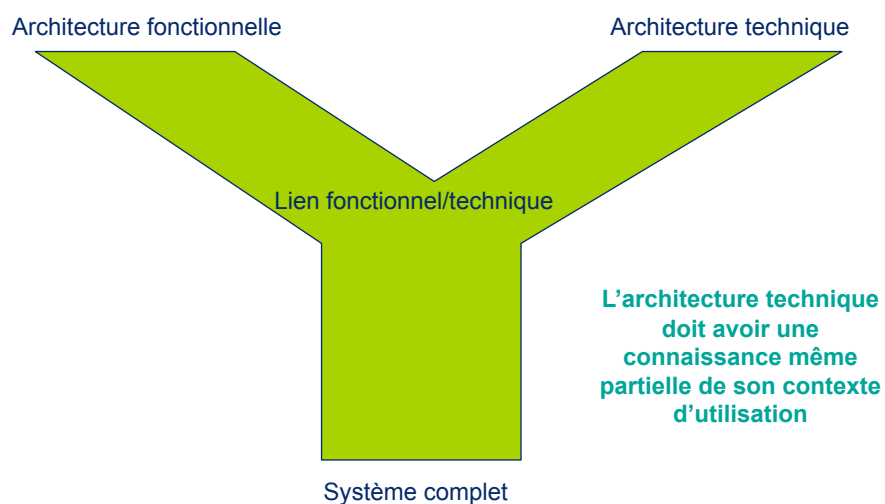
Génération de code

113



insertion de préoccupations

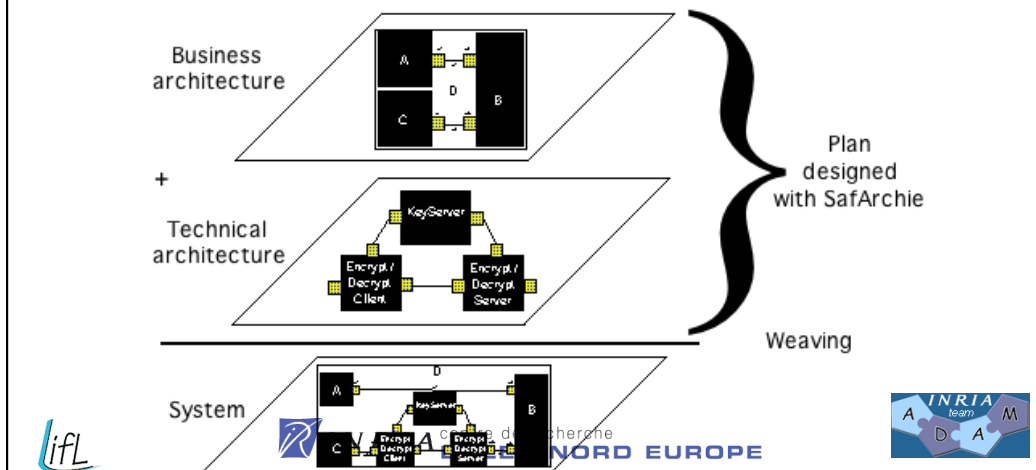
114



A unified transformation approach

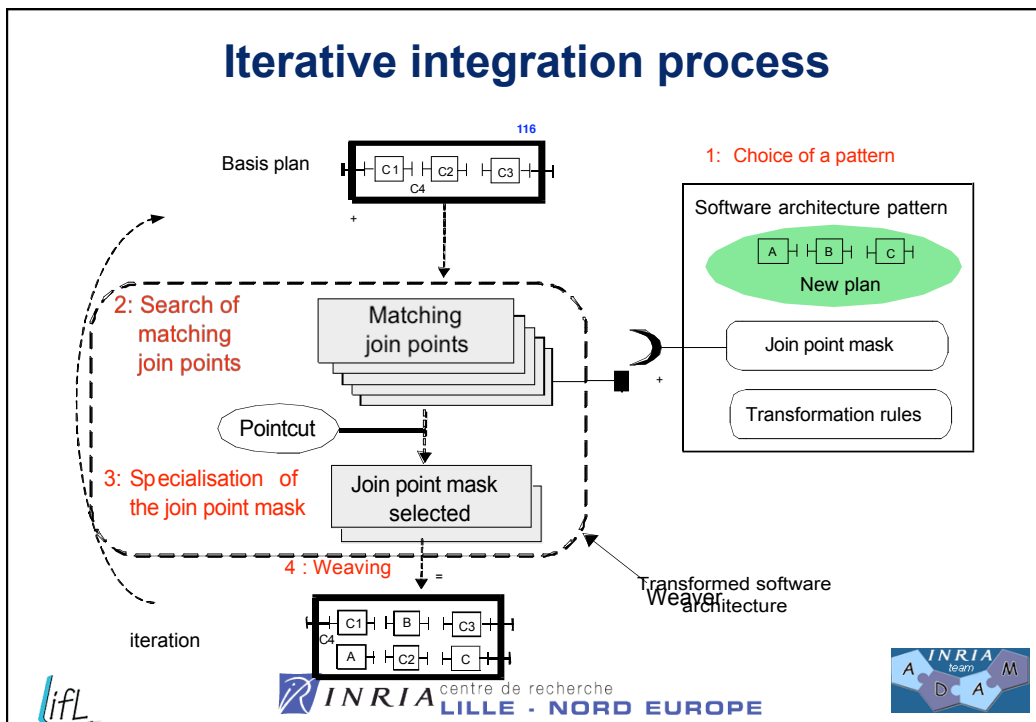
115

Building a software architecture by superposing plans



Iterative integration process

116



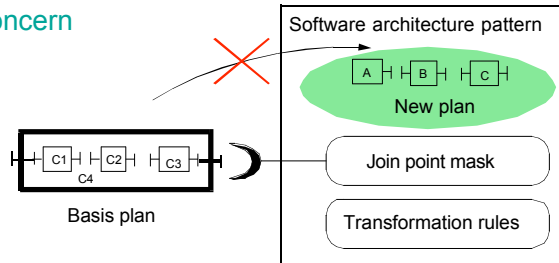
The software architecture pattern

117

A new entity to modularize a concern

Respect the dependency inversion principle

Inspired by AOP



TranSAT

AOP

Basis plan

Basis program

Software architecture pattern

Aspect

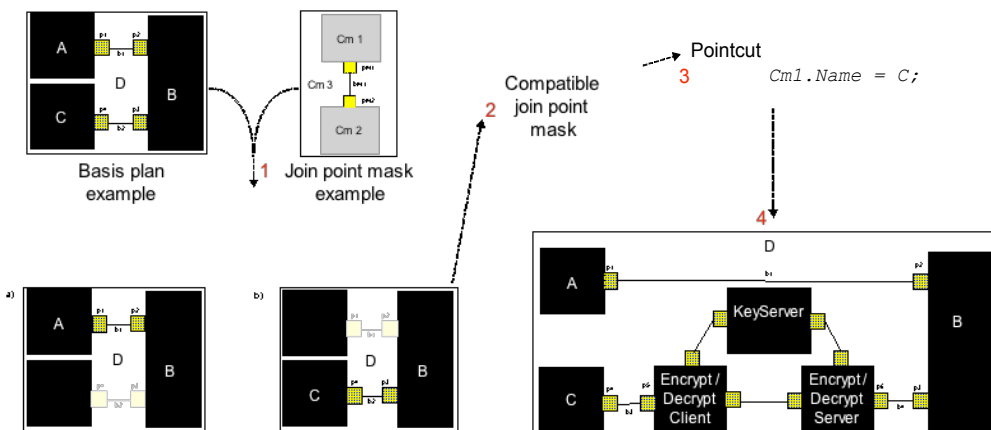
- ☞ new plan of architecture
- ☞ join point mask
- ☞ transformation rules

- ☞ advice
- ☞ join point type (static)
- ☞ weaving (implicit)



TranSAT process

118



VI. Conclusion



Conclusion

Les ADLs sont des langages de modélisation

- ☞ prédominance d'artefacts pour l'assemblage dès la phase de modélisation

Tentatives de combiner UML avec les ADLs

- ☞ « *Reconciling the Needs of Architectural Description with Object-Modeling Notations* » David Garlan and Andrew J. Kompanek

Tentative de définir une taxinomie de connecteurs

- ☞ « *Towards a taxonomy of Software Connectors* », Nenad Medvidovic, Mehta, Phadke



Conclusion

121

Description de l'architecture en intégrant

- ☞ structuration hiérarchique des composants logiciels
- ☞ spécification externalisée de la communication
 - ➔ *Connecteurs*
- ☞ spécification d'une partie du comportement dynamique
 - ➔ *Instantiation/suppression de composants*
 - Collection
 - ➔ *Communication*
 - Désignation associative

Modèle d'intégration de description

- ☞ Traitement de l'hétérogénéité des types de logiciels

Modèle abstrait pour l'évolution du logiciel



Bibliographie

122

Livres

- Component Software: Beyond Object-Oriented Programming de Clemens Szyperski, Stephan Murer, Dominik Gruntz - 450 pages 2nd édition (novembre 2002) Addison Wesley Longman; ISBN : 0201745720
- Building Reliable Component-Based Software Systems by Ivica Crnkovic (Editor), Magnus Larsson (Editor) - 539 pages (juillet 2002) Artech House Inc; ISBN : 1580533272
- Pattern-Oriented Software Architecture A System of Patterns, F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal John Wileyand Sons Ltd, Chichester, UK, 1996 ISBN 0-471-95869-7
- Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach ISBN 0-201-67494-7, May 2000. Jan Bosch, Addison Wesley.
- Model Driven Architecture: Applying MDA to Enterprise Computing. FrankelL - John Wiley. ISBN : 0-471-31920-1 (02/2003) 327 p
-



Bibliographie

123

Livres

- MDA explained: The Practice and Promise of the Model Driven Architecture. Kleppe - Longman Paul ISBN : 0-321-19442-X (06/2003) 170 p.
- The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition by Jos Warmer, Anneke Kleppe Publisher: Addison-Wesley Pub Co; 2nd edition (August 29, 2003) ISBN: 0321179366
- The Unified Modeling Language Reference Manual, J. Rumbaugh, I. Jacobson, G. Booch, Addison Wesley, 1999
- UML Distilled, M. Fowler, Addison Wesley, 1998
- La modélisation avec UML, Muller, Eyrolles, 3ème édition, 1999
- UML Components, J. Cheesman, J. Daniels, Addison Wesley, 2001
- Object, Components and Framework with UML, The Catalysis Approach, D. D'Souza, A. Willis, Addison Wesley, 1999



Bibliographie

124

Des articles de fond sur les ADLs

<http://www.sti.uniurb.it/events/sfm03sa/programme.html> (école d'été 2003)

- D. Perry and A. Wolf, Foundations for the study of Software Architecture, ACM Sigsoft Software Eng. Notes 17(4), oct 92
- D. Garlan and D. Parry, Special issue on Software Architecture, IEEE TSE 21(4) avr95
- M. Shaw and D. Garlan, Software Architecture, perspectives on an emerging discipline Prentice Hall, 1996,
- L. Bass, P. Clements, R. Kazman, Software architecture in practice, Addison Wesley, 98
- D. Garlan Software architecture : a roadmap dans The Future Software Engineering, A. Finkelstein IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471-2000
- P. Kruchten
The 4.1 view model of architecture, IEEE Software 12(6) nov 95
- N. Medvidovic and R. Taylor A classification and comparison framework for software architecture description languages



Bibliographie

125

Les différents ADLs

SafArchie/Transat : <http://transat.gforge.inria.fr/>

ACME: <http://www.cs.cmu.edu/~acme>

Rapide: <http://pavg.stanford.edu/rapide/>

Wright: <http://www.cs.cmu.edu/afs/cs/project/able/www/wright/index.html>

Aesop: http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html

UniCon: <http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/index.html>

C2 SADL: <http://www.ics.uci.edu/pub/arch/>

SSEP: <http://www.mcc.com/projects/ssepp>

ADML: <http://www.mcc.com/projects/ssepp/adml>

ArchJava : <http://www.cs.washington.edu/homes/jonal/archjava/>

